第20章

数据库函数库

20.1 引言

20世纪80年代早期，UNIX系统被认为不是一个适合运行多用户数据库系统的环境（见Stonebraker[1981]和Weinberger[1982]）。早期的系统（如V7），因为没有提供任何形式的IPC机制（除了半双工管道），也没有提供任何形式的字节范围锁机制，所以确实不适合运行多用户数据库系统。但是，这些缺陷中的大多数都已得到纠正。到了20世纪80年代后期，UNIX系统已为运行可靠的、多用户的数据库系统提供了一个适合的环境。自那时以来很多商业公司都已提供这种数据库系统。

本章将开发一个简单的、多用户数据库的C函数库。调用此函数库提供的C语言函数，其他程序可以读取和存储数据库中的记录。这个C函数库通常只是一个完整的数据库系统的一部分，这里并不开发其他部分（如查询语言等），关于其他部分可以参阅专门介绍数据库系统的教科书。我们感兴趣的是数据库函数库与UNIX系统的接口，以及这些接口与前面各章节所涉及主题的关系（如14.3节的字节范围锁）。

20.2 历史

dbm(3)是在UNIX系统中很流行的数据库函数库，它由Ken Thompson开发，使用了动态散列结构。最初，它与V7一起提供，并出现在所有BSD版本中，也包含在SVR4的BSD兼容函数库中[AT&T 1990c]。BSD的开发者扩充了dbm函数库，并将它称为ndbm。ndbm函数库包括在BSD和SVR4中。ndbm函数被标准化后成为Single UNIX Specification的XSI扩展部分。

Seltzer和Yigit[1991]中详细介绍了dbm函数库使用的动态散列算法的历史，以及这个库的其他实现方法（例如，dbm函数库的GNU版本gdbm）。但是，所有这些实现的一个根本缺点是：它们都不支持多个进程对数据库的并发更新。它们都没有提供并发控制（如记录锁）。

4.4BSD提供了一个新的库——db(3)，该库支持三种不同的访问模式：面向记录、散列和B-树。同样，db(3)也没有提供并发控制（这一点在db(3)手册页的BUGS部分中说得很清楚）。

Sleepycat Software（http://www.sleepycat.com）提供了几个db函数库版本，它们支持并发访问、锁和事务。

绝大部分商用数据库函数库提供多进程同时更新数据库所需要的并发控制。这些系统一般都使用14.3节中介绍的建议记录锁，但是，它们也常常实现自己的锁原语，以避免为获得一把无竞争的锁而需的系统调用开销。这些商用系统通常用B+树[Comer 1979]，或者某种动态散列技术（例如线性散列[Litwin 1980]或者可扩展的散列[Fagin et al. 1979]）来实现数据库。

表20-1列出了本书说明的四种操作系统中常用的数据库函数库。注意在Linux上，gdbm库既支持dbm函数库，又支持ndbm函数库。

表20-1　多种平台支持的数据库函数库

函数库	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
dbm	XSI		gdbm		•
ndbm		•	gdbm	•	•
db		•	•	•	•

20.3　函数库

本章开发的函数库类似于ndbm函数库，但增加了并发控制机制，从而允许多进程同时更新同一数据库。本节将进述数据库函数库的C语言接口，下一节再讨论其实际的实现。

当打开数据库时，通过返回值得到代表数据库的句柄（一个难以理解的指针（opaque pointer））。此句柄将作为参数传递给其他数据库函数。

```
#include "apue_db.h"
DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);
                                        返回值：若成功则返回数据库句柄，若出错则返回NULL
void db_close(DBHANDLE db);
```

710

如果db_open成功返回，则将建立两个文件：*pathname.idx*和*pathname.dat*，*pathname.idx*是索引文件，*pathname.dat*是数据文件。open的第二个参数*oflag*（见3.3节）指定这些文件的打开模式（只读、读/写或如果文件不存在则创建等）。如果需要建立新的数据库文件，*mode*将作为第三个参数传递给open（文件访问权限）。

当不再使用数据库时，调用db_close来关闭数据库。db_close将关闭索引文件和数据文件，并释放数据库使用过程中分配到的所有用于内部缓冲的存储空间。

当向数据库中加入一条新的记录时，必须指明此记录的键，以及与此键相联系的数据。如果此数据库存储的是人事信息，键可以是雇员ID号，数据可以是此雇员的姓名、地址、电话号码以及受聘日期等等。我们的实现要求每条记录的键必须是唯一的（例如，两条雇员记录不能有同样的雇员ID号）。

```
#include "apue_db.h"
int db_store(DBHANDLE db, const char *key, const char *data,
             int flag);
                                        返回值：若成功则返回0，若出错则返回非0值（见下）
```

参数*key*和*data*是由null结束的字符串。它们可以包含除了null外的任何字符，如换行符。

*flag*参数只能是DB_INSERT（加一条新记录）、DB_REPLACE（替换一条已有的记录）或DB_STORE（加一条新记录或替换一条已有的记录，只要合适无论哪一种都可以）。这三个常数定义在apue_db.h头文件中。如果使用DB_INSERT或DB_STORE，并且记录并不存在，则加入一条新记录；如果使用DB_REPLACE或DB_STORE，并且该记录已经存在，则用新记录替

换已存在的原记录；如果使用DB_REPLACE，而记录不存在，则errno设置为ENOENT，返回值为−1，并且不加入新记录；如果使用DB_INSERT，而记录已经存在，则不加入新记录，返回值为1，在这里，返回1以区别于一般的出错返回（−1）。

通过提供键*key*可以从数据库中取出一条记录。

```
#include "apue_db.h"
char *db_fetch(DBHANDLE db, const char *key);
                        返回值：若成功则返回指向数据的指针，若记录没有找到则返回NULL
```

如果记录找到了，则返回指向按键*key*存放的数据的指针。通过指明键*key*，也可以在数据库中删除一条记录。

```
#include "apue_db.h"
int db_delete(DBHANDLE db, const char *key);
                        返回值：若成功则返回0，若记录没有找到则返回−1
```

711

除了通过键访问数据库外，也可以一条一条记录地访问数据库。为此，首先调用db_rewind回滚到数据库的第一条记录，然后在每一次循环中调用db_nextrec，顺序地读每条记录。

```
#include "apue_db.h"
void db_rewind(DBHANDLE db);
char *db_nextrec(DBHANDLE db, char *key);
                        返回值：若成功则返回指向数据的指针，若到达数据库的结尾则返回NULL
```

如果*key*是非空的指针，则db_nextrec将当前记录的键复制到*key*所指向的存储区中。

db_nextrec不保证记录访问的次序，只保证每一条记录被访问恰好一次。纵使顺序地存储三条键分别为A、B、C的记录，也无法确定db_nextrec将按什么顺序返回这三条记录。它可能按B、A、C的顺序返回，也可能按其他顺序，实际的顺序由数据库的实现决定。

这七个函数提供了数据库函数库的接口。接下来介绍实现。

20.4 实现概述

大多数访问数据库的函数库使用两个文件来存储信息：一个索引文件和一个数据文件。索引文件包含索引值（键）和指向数据文件中对应数据记录的指针。有许多技术可用来组织索引文件，以提高按键查询的速度和效率，散列法和B+树是两种常用的技术。我们采用固定大小的散列表来组织索引文件结构，并采用链表法解决散列冲突。在介绍db_open时，曾提到将创建两个文件：一个以.idx为后缀的索引文件和一个以.dat为后缀的数据文件。

这里将键和索引以由null结尾的字符串形式存储——它们不能包含任意的二进制数据。有些数据库系统用二进制形式存储数值数据（例如，用1、2或4个字节存储一个整数）以节省存储空间，这样一来使函数复杂化，也使数据库文件在不同的平台间移植比较困难。例如，网络上有两个系统使用不同的二进制格式存储整数，如果想要这两个系统都能够访问数据库就必须解决不同存储格式的问题（今天不同体系结构的系统共享文件已经很常见了）。按照字符串形式

存储所有的记录（包括键和数据）能使这一切变得简单，尽管这确实需要使用更多的磁盘空间，但这是为可移植性付出的很小代价。

db_store要求对于每个键，只有一条对应的记录。有些数据库系统允许多条记录使用同样的键，并提供方法访问与一个键相关的所有记录。另外，由于只有一个索引文件，这意味着
712 每条数据记录只能有一个键（不支持第二个键）。有些数据库系统允许一条记录拥有多个键，并且对每一个键使用一个索引文件，当加入或删除一条记录时，要对所有的索引文件进行相应的更新。（一个有多个索引文件的例子是雇员库文件，可以将雇员ID号作为索引的键，也可以将雇员的社会保险号作为索引键。由于雇员的名字并不保证唯一，所以名字不能作为键。）

图20-1是数据库实现的基本结构。索引文件由三部分组成：空闲链表指针、散列表和索引记录。在图20-1中，所有*ptr*字段中实际存储的是以ASCII码数字形式记录的文件中的偏移量。

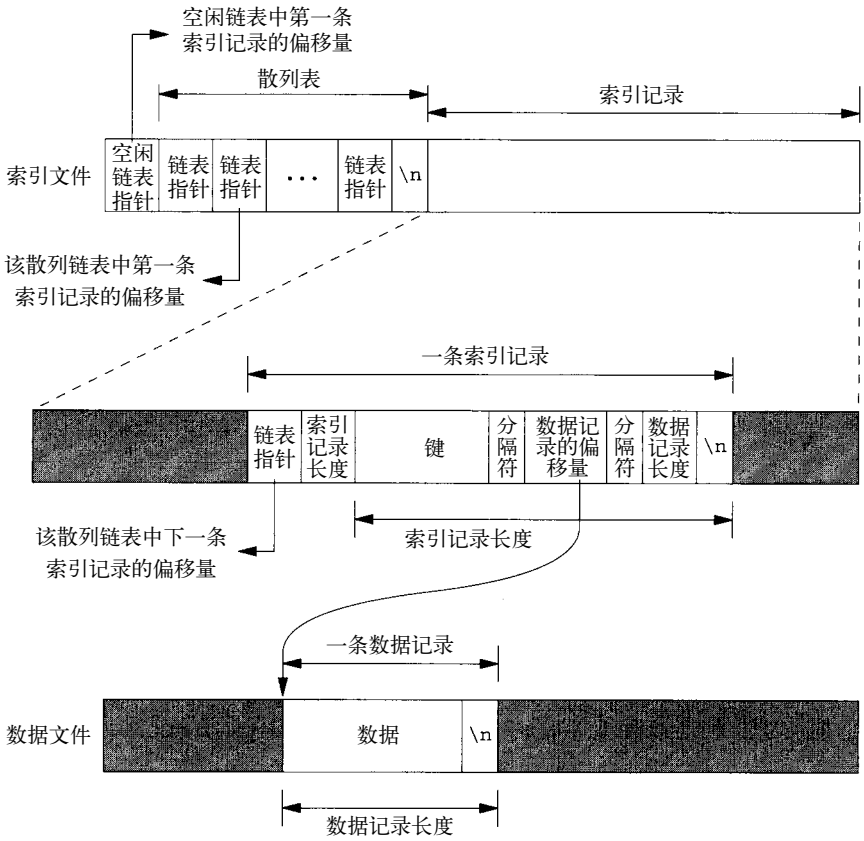

图20-1　索引文件和数据文件结构

当给定一个键要在数据库中寻找一条记录时，db_fetch根据该键计算散列值，由此散列
713 值可确定散列表中的一条散列链（链表指针字段可以为0，表示一条空的散列链）。沿着这条散列链，可以找到所有具有该散列值的索引记录。当遇到一条索引记录的链表指针字段为0时，表示到达了此散列链的末尾。

下面来看一个实际的数据库文件。程序清单20-1中的程序建立了一个新的数据库，并且写入了三条记录。由于所有的字段都以ASCII字符的形式存储在数据库中，所以可以用任何标准的UNIX系统工具来查看实际的索引文件和数据文件。

```
$ ls -l db4.*
-rw-r--r--  1 sar        28 Oct 19 21:33 db4.dat
-rw-r--r--  1 sar        72 Oct 19 21:33 db4.idx
$ cat db4.idx
   0  53  35   0
   0  10Alpha:0:6
   0  10beta:6:14
  17  11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

为了使这个例子简单，将每个指针（*ptr*）字段的大小定为4个ASCII字符，将散列链的条数定为3。由于每一个*ptr*记录的是一个文件偏移量，所以4个ASCII字符限制了一个索引文件或数据文件的大小最多只能为10 000字节。当在20.9节做数据库系统的性能测试时，将*ptr*字段的大小设为6个字符（这样文件大小可以达到1 000 000字节），将散列链数设为100以上。

程序清单20-1 建立一个数据库并向其中写三条记录

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

索引文件的第一行为：

```
0  53  35   0
```

分别为空闲链表指针（0表示空闲链表为空），和三个散列链的指针：53、35和0。下一行：

```
0  10Alpha:0:6
```

显示了一条索引记录的结构。第一个4字符字段（0）为链表指针，表示这一条记录是此散列链的最后一条。下一个4字符字段（10）为索引记录长度（*idx len*），表示此索引记录剩余部分的长度。用两个`read`操作来读取一条索引记录：第一个`read`读取这两个固定长度的字段（*chain ptr*和*idx len*），然后再根据*idx len*来读取后面的不定长部分。剩下的三个字段为：键（*key*）、数据记录的偏移量（*dat off*）和数据记录的长度（*dat len*），这三个字段用分隔符（*sep*）隔开，在这里使用的是冒号。由于此三个字段都是不定长的，所以需要一个专门的分隔符，而且这个分隔符不能出现在键中。最后用一个`\n`（换行符）结束这一条索引记录。由于在`idx len`中已经

有了记录的长度，所以这个换行符并不是必需的，加上换行符是为了把各条索引记录分开，这样就可以用标准的UNIX系统工具（如cat和more）来查看索引文件。键（*key*）是将记录加入数据库时指定的值。数据记录在数据文件中的偏移量为0，长度为6。从数据文件中可看到数据记录确实从0开始，长度为6个字节。（与索引文件一样，这里自动在每条数据记录的后面加上一个换行符，以便于使用UNIX系统工具。在调用db_fetch时，此换行符不作为数据返回。）

如果在这个例子中跟踪三条散列链，可以看到第一条散列链上第一条记录的偏移量是53（gamma），这条链上下一条记录的偏移量为17（alpha），并且是这条链上的最后一条记录。第二条散列链上的第一条记录的偏移量是35（beta），且是此链上最后一条记录。第三条散列链为空。

请注意索引文件中键的顺序和数据文件中对应数据记录的顺序与程序清单20-1的程序中调用db_store的顺序相同。由于在调用db_open时使用了O_TRUNC标志，索引文件和数据文件都被截断，整个数据库相当于重新初始化。在这种情形下，db_store将新的索引记录和数据记录添加到对应的文件末尾。后面将看到db_store也可以重复使用这两个文件中因删除记录而生成的空间。

在这里使用固定大小的散列表作为索引是一个折衷，当每个散列链都不太长时，这种方法能保证快速地查找。我们的目的是能够快速地查找任一键，同时又不使用太复杂的数据结构，如B-树或动态可扩充散列表。动态可扩充散列表的优点是能保证仅用两次磁盘操作就能找到数据记录（详见Litwin [1980] 或 Fagin et al. [1979]）。B-树能够用（已排序的）键的顺序来遍历数据库（采用散列表的db_nextrec函数就做不到这一点）。

714
~
715

20.5　集中式或非集中式

当有多个进程访问同一数据库时，有两种方法可实现库函数：

(1) 集中式。由一个进程作为数据库管理者，所有的数据库访问工作由此进程完成。其他进程通过IPC机制与此中心进程进行联系。

(2) 非集中式。每个库函数独立申请并发控制（加锁），然后自己调用I/O函数。

使用这两种技术的数据库系统都有。如果有适当的加锁例程，因为避免了使用IPC，那么非集中式方法一般要快一些。图20-2描绘了集中式方法的操作。

图中特意表示出IPC像绝大多数UNIX系统的消息传送一样需要经过操作系统内核（15.9节中说明的共享存储不需要这种经过内核的复制）。可以看出，在集中式方法下，中心控制进程将记录读出，然后通过IPC机制将数据传送给请求进程，这是这种设计的不足之处。注意，集中式数据库管理进程是唯一对数据库文件进行I/O操作的进程。

集中式的优点是能够根据需要来对操作模式进行调整。例如，可以通过中心进程给不同的进程赋予不同的优先级，这会影响到中心进程对I/O操作的调度。而用非集中式方法则很难做到这一点。在这种情况下只能依赖于操作系统内核的磁盘I/O调度策略和加锁策略（也就是说，当三个进程同时等待一个锁开锁时，哪个进程下一个得到锁？）。

集中式方法的另一个优点是，复原要比非集中式方法容易。在集中式方法中，所有状态信息都集中存放在一处，所以如若杀死了数据库进程，那么只需在该处查看以识别出需要解决的未完成事务，然后将数据库恢复到一致状态。

图20-3描绘了非集中式方法，本章的实现就是采用这种方法。

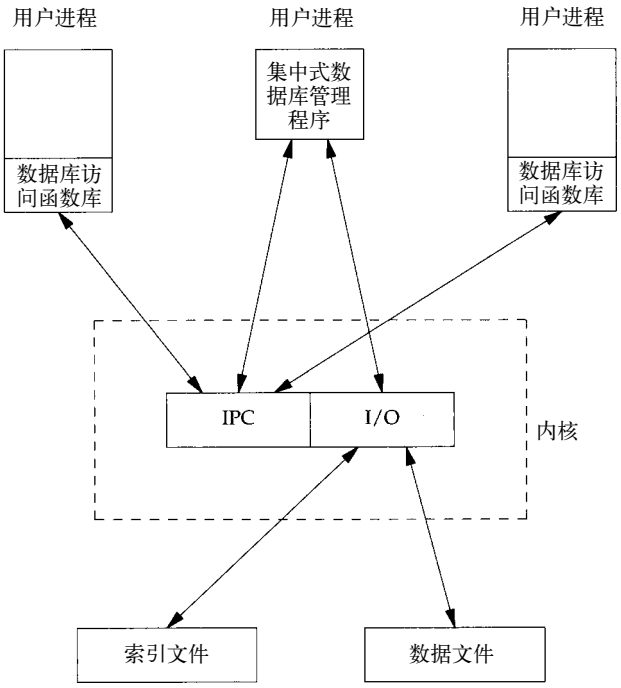

图20-2 集中式数据库访问

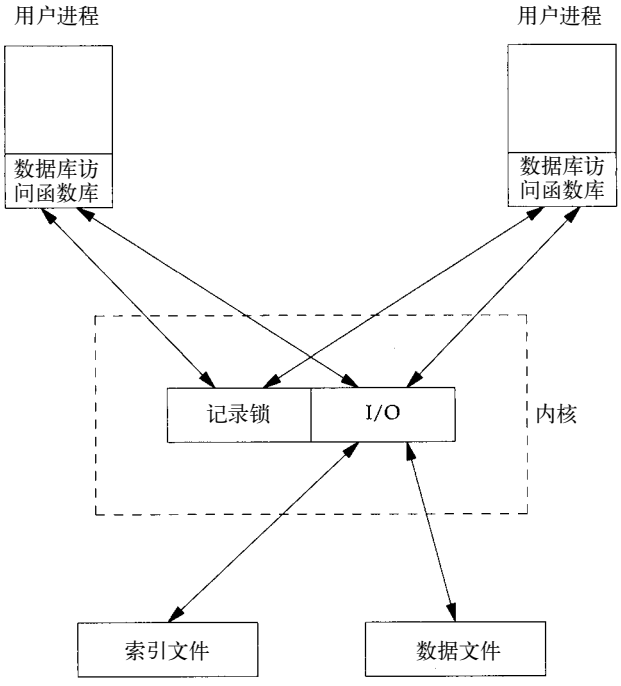

图20-3 非集中式数据库访问

调用数据库库函数执行I/O操作的用户进程是合作进程，它们使用字节范围锁机制来实现并发访问控制。

716 ~ 717

20.6 并发

由于很多系统的实现都采用两个文件（一个索引文件和一个数据文件）的方式，所以这里也特地使用这种实现方法，这要求能够控制对两个文件的加锁。有很多方法可用来对两个文件进行加锁。

1. 粗锁

最简单的加锁方法是将这两个文件中的一个作为整个数据库的锁，并要求调用者在对数据库进行操作前必须获得这个锁，这种加锁方式称为*粗锁*（coarse-grained locking）。例如，可以认为一个进程对索引文件的0字节加了读锁后，就能读整个数据库；一个进程对索引文件的0字节加了写锁后，就能写整个数据库。可以使用UNIX系统的字节范围锁机制来控制每次可以有多个读进程，而只能有一个写进程（见表14-2）。`db_fetch`和`db_nextrec`函数将要求具有读锁，而`db_delete`、`db_store`以及`db_open`则要求具有写锁。（`db_open`要求写锁的原因是：如果要创建新文件的话，要在索引文件前端建立空的空闲链表以及散列链表。）

粗锁的问题是它限制了最大程度的并发。用粗锁时，当一个进程向一条散列链中加入一条记录时，其他进程无法访问另一条散列链上的记录。

2. 细锁

下面用称为*细锁*（fine-grained locking）的方法来改进粗锁以提高并发度。首先，要求一个读进程或写进程在操作一条记录前必须先获得此记录所在散列链的读锁或写锁，允许对同一条散列链同时可以有多个读进程，而只能有一个写进程。其次，一个写进程在操作空闲链表（如`db_delete`或`db_store`）前，必须获得空闲链表的写锁。最后，当`db_store`向索引文件或数据文件末尾追加一条新记录时，必须获得对应文件相应区域的写锁。

期望细锁能比粗锁提供更高的并发度，20.9节将给出一些实际的比较测试结果。20.8节给出了采用细锁实现的源代码，并详细讨论了锁的实现（粗锁是实现的简化）。

在源代码中，直接调用了`read`，`readv`，`write`和`writev`，而没有使用标准I/O函数库。虽然使用标准I/O函数库也可以使用字节范围锁，但是需要非常复杂的缓冲管理。例如，当另一
718 个进程在5分钟之前修改了数据，不希望`fgets`返回10分钟前读入标准I/O缓冲的数据。

这里对并发进行讨论，所依据的是对数据库函数库的简单需求，商业系统一般有更多的需要。关于并发更多的细节可以参见Date[2004]的第16章。

20.7 构造函数库

数据库的函数库由两个文件构成，它们是：公用的C头文件以及一个C源文件。可以用下列命令构造一静态函数库。

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

因为在数据库函数库中使用了一些我们自己的公共函数，所以希望与`libapue_db.a`相链接的应用程序也需要与`libapue.a`相链接。

另一方面，如果想构建数据库函数库的动态共享库版本，那么可使用下列命令：

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

构建成的共享库libapue_db.so.1需放置在动态链接/装入程序（dynamic linker/loader）能够找到的公用目录中。另一个替代的方法是：将共享库放置在私有目录中，修改LD_LIBRARY_PATH环境变量，使动态链接/装入程序的搜索路径包含该私有目录。

在不同平台上构建共享库的步骤会有所不同。这里所给出的步骤是在带GNU C编译器的Linux系统中进行的。

20.8 源代码

本节对所编写的数据库函数库源代码进行解说，先从头文件apue_db.h开始。函数库源代码以及调用此函数库的所有应用程序都包含这一头文件。

从此处开始，实例程序的编排方式在很多方面与前面的有所不同。首先，因为源代码较长，为此加了行号，这使得联系相应的源代码进行讨论更加方便。其次，对源代码的说明紧随其后。

这种风格受到John Lions对UNIX V6源代码注释一书[Lions 1977，1996]的影响，使得解释说明大量源代码更为简易。

注意，这里对空白行不编号。虽然某些工具（例如pr(1)）的正常操作与这些空白行是有关的，但是此处对它们并无任何兴趣。 719

```
 1  #ifndef _APUE_DB_H
 2  #define _APUE_DB_H

 3  typedef    void *  DBHANDLE;

 4  DBHANDLE  db_open(const char *, int, ...);
 5  void      db_close(DBHANDLE);
 6  char     *db_fetch(DBHANDLE, const char *);
 7  int       db_store(DBHANDLE, const char *, const char *, int);
 8  int       db_delete(DBHANDLE, const char *);
 9  void      db_rewind(DBHANDLE);
10  char     *db_nextrec(DBHANDLE, char *);

11  /*
12   * Flags for db_store().
13   */
14  #define DB_INSERT     1    /* insert new record only */
15  #define DB_REPLACE    2    /* replace existing record */
16  #define DB_STORE      3    /* replace or insert */

17  /*
18   * Implementation limits.
19   */
20  #define IDXLEN_MIN    6    /* key, sep, start, sep, length, \n */
21  #define IDXLEN_MAX 1024    /* arbitrary */
22  #define DATLEN_MIN    2    /* data byte, newline */
23  #define DATLEN_MAX 1024    /* arbitrary */

24  #endif /* _APUE_DB_H */
```

[1-3] 使用符号_APUE_DB_H以保证只包含该头文件一次。DBHANDLE类型表示对数据库的一个有效引用，用于隔离应用程序和数据库的实现细节。数据库函数库

向应用程序提供DBHANDLE类型，而标准I/O库则向应用程序提供FILE结构，两者相比非常相似。

[4-10] 接着，声明了数据库函数库公用函数的原型。因为使用函数库的应用程序包含了此文件，所以这里不再声明函数库私有函数的原型。

[11-24] 定义了可以传送给db_store函数的合法标志，其后是实现的基本限制。为支持更大的数据库可以更改这些限制。

最小索引记录长度由IDXLEN_MIN指定。这表示1字节键、1字节分隔符、1字节起始偏移量、另一个1字节分隔符、1字节长度和终止换行符。（回忆图20-1中索引记录的格式。）一条索引记录通常长于IDXLEN_MIN字节，这只是最小长度。

720

下一个文件是db.c，它是库函数的C源文件。为简化起见，将所有函数都放在一个文件中。这样处理的优点是只要将私有函数声明为static，就可对外将它隐蔽起来。

```
 1  #include "apue.h"
 2  #include "apue_db.h"
 3  #include <fcntl.h>      /* open & db_open flags */
 4  #include <stdarg.h>
 5  #include <errno.h>
 6  #include <sys/uio.h>    /* struct iovec */

 7  /*
 8   * Internal index file constants.
 9   * These are used to construct records in the
10   * index file and data file.
11   */
12  #define IDXLEN_SZ     4    /* index record length (ASCII chars) */
13  #define SEP         ':'    /* separator char in index record */
14  #define SPACE       ' '    /* space character */
15  #define NEWLINE     '\n'   /* newline character */

16  /*
17   * The following definitions are for hash chains and free
18   * list chain in the index file.
19   */
20  #define PTR_SZ        6    /* size of ptr field in hash chain */
21  #define PTR_MAX  999999    /* max file offset = 10**PTR_SZ - 1 */
22  #define NHASH_DEF   137    /* default hash table size */
23  #define FREE_OFF      0    /* free list offset in index file */
24  #define HASH_OFF PTR_SZ    /* hash table offset in index file */

25  typedef unsigned long  DBHASH; /* hash values */
26  typedef unsigned long  COUNT;  /* unsigned counter */
```

[1-6] 由于使用了一些私有函数库中的函数，所以程序中包含了apue.h。当然，apue.h也包含若干标准头文件，包括<stdio.h>和<unistd.h>。因为db_open函数使用由<stdarg.h>定义的可变参数函数，所以程序中也包含了<stdarg.h>。

[7-26] 索引记录的长度指定为IDXLEN_SZ。用某些字符（例如冒号、换行符）作为数据库中的分隔符。当删除一条记录时，在其中全部填入空格符。

其中一些定义为常量的值也可定义为变量，只是这样会使实现复杂一些。例如，设定散列表的大小为137记录项，也许更好的方法是让db_open的调用者根据预期的数据库大小通过参数来设定这个值，然后将该值存储在索引文件的最前面。

721

```
27  /*
28   * Library's private representation of the database.
29   */
30  typedef struct {
31    int    idxfd;  /* fd for index file */
32    int    datfd;  /* fd for data file */
33    char  *idxbuf; /* malloc'ed buffer for index record */
34    char  *datbuf; /* malloc'ed buffer for data record*/
35    char  *name;   /* name db was opened under */
36    off_t  idxoff; /* offset in index file of index record */
37                   /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
38    size_t idxlen; /* length of index record */
39                   /* excludes IDXLEN_SZ bytes at front of record */
40                   /* includes newline at end of index record */
41    off_t  datoff; /* offset in data file of data record */
42    size_t datlen; /* length of data record */
43                   /* includes newline at end */
44    off_t  ptrval; /* contents of chain ptr in index record */
45    off_t  ptroff; /* chain ptr offset pointing to this idx record */
46    off_t  chainoff; /* offset of hash chain for this index record */
47    off_t  hashoff;  /* offset in index file of hash table */
48    DBHASH nhash;    /* current hash table size */
49    COUNT  cnt_delok;    /* delete OK */
50    COUNT  cnt_delerr;   /* delete error */
51    COUNT  cnt_fetchok;  /* fetch OK */
52    COUNT  cnt_fetcherr; /* fetch error */
53    COUNT  cnt_nextrec;  /* nextrec */
54    COUNT  cnt_stor1;    /* store: DB_INSERT, no empty, appended */
55    COUNT  cnt_stor2;    /* store: DB_INSERT, found empty, reused */
56    COUNT  cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
57    COUNT  cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
58    COUNT  cnt_storerr;  /* store error */
59  } DB;
```

[27-48] 在DB结构中记录一个打开数据库的所有信息。db_open函数返回DB结构的指针DBHANDLE值，这个指针被用于其他所有函数，而该结构本身则不面向调用者。因为在数据库中是以ASCII码形式存放指针和长度，所以要将这些ASCII码变换为数字值，然后存放在DB结构中。DB结构中也存放散列表长度，虽然一般而言，这是定长的，但也有可能为增强该函数库，允许调用者在创建数据库时指定该长度（见习题20.7）。

[49-59] DB结构的最后10个字段对成功和不成功的操作计数。如果想要分析数据库的性能，则可编写一个函数返回这些统计值。但目前仅保持这些计数器，并未编写此种函数。

722

```
60  /*
61   * Internal functions.
62   */
63  static DB     *_db_alloc(int);
64  static void    _db_dodelete(DB *);
65  static int     _db_find_and_lock(DB *, const char *, int);
66  static int     _db_findfree(DB *, int, int);
67  static void    _db_free(DB *);
68  static DBHASH  _db_hash(DB *, const char *);
69  static char   *_db_readdat(DB *);
70  static off_t   _db_readidx(DB *, off_t);
```

```
71  static off_t   _db_readptr(DB *, off_t);
72  static void    _db_writedat(DB *, const char *, off_t, int);
73  static void    _db_writeidx(DB *, const char *, off_t, int, off_t);
74  static void    _db_writeptr(DB *, off_t, off_t);

75  /*
76   * Open or create a database.  Same arguments as open(2).
77   */
78  DBHANDLE
79  db_open(const char *pathname, int oflag, ...)
80  {
81      DB          *db;
82      int         len, mode;
83      size_t      i;
84      char        asciiptr[PTR_SZ + 1],
85                  hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                      /* +2 for newline and null */
87      struct stat statbuff;

88      /*
89       * Allocate a DB structure, and the buffers it needs.
90       */
91      len = strlen(pathname);
92      if ((db = _db_alloc(len)) == NULL)
93          err_dump("db_open: _db_alloc error for DB");
```

[60-74]　选择用db_开头来命名所有用户可调用（公用）的库函数，用_db_开头来命名内部（私有）函数。公用函数在函数库头文件apue_db.h中声明。内部函数声明为static，所以只有同一文件中的其他函数才能调用它们（该文件包含函数库实现）。

[75-93]　db_open函数的参数与open(2)相同。如果调用者想要创建数据库文件，那么用可选择的第三个参数指定文件权限。db_open函数打开索引文件和数据文件，在必要时初始化索引文件。该函数调用_db_alloc来为DB结构分配空间，并初始化此结构。

723

```
94      db->nhash    = NHASH_DEF;/* hash table size */
95      db->hashoff = HASH_OFF; /* offset in index file of hash table */
96      strcpy(db->name, pathname);
97      strcat(db->name, ".idx");

98      if (oflag & O_CREAT) {
99          va_list ap;

100         va_start(ap, oflag);
101         mode = va_arg(ap, int);
102         va_end(ap);

103         /*
104          * Open index file and data file.
105          */
106         db->idxfd = open(db->name, oflag, mode);
107         strcpy(db->name + len, ".dat");
108         db->datfd = open(db->name, oflag, mode);
109     } else {
110         /*
111          * Open index file and data file.
112          */
113         db->idxfd = open(db->name, oflag);
```

```
114         strcpy(db->name + len, ".dat");
115         db->datfd = open(db->name, oflag);
116     }

117     if (db->idxfd < 0 || db->datfd < 0) {
118         _db_free(db);
119         return(NULL);
120     }
```

[94-97] 继续初始化DB结构。调用者传入的路径名指定数据库文件名的前缀。添加后缀.idx以构成数据库索引文件的名字。

[98-108] 如果调用者想要创建数据库文件，那么使用<stdarg.h>中的可变参数函数以找到可选的第3个参数，然后，使用open来创建和打开索引文件和数据库文件。注意，数据文件的文件名与索引文件以同样的前缀开始，只是后缀为.dat。

[109-116] 如果调用者没有指定O_CREAT标志，那么正在打开现有的数据库文件。此时，只用两个参数调用open。

[117-120] 如果在打开或创建任一数据库文件时出错，则调用_db_free清除DB结构，然后对调用者返回NULL。如果一个文件open成功，另一个失败，_db_free将关闭该打开的文件描述符。很快就会见到这一操作。 724

```
121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123          * If the database was created, we have to initialize
124          * it.  Write lock the entire file so that we can stat
125          * it, check its size, and initialize it, atomically.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: writew_lock error");

129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: fstat error");

131         if (statbuff.st_size == 0) {
132             /*
133              * We have to build a list of (NHASH_DEF + 1) chain
134              * ptrs with a value of 0.  The +1 is for the free
135              * list pointer that precedes the hash table.
136              */
137             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121-130] 如果正在建立数据库，则必须正确地加锁。考虑两个进程试图同时建立同一个数据库的情况。假设第一个进程运行到调用fstat，并且在fstat返回后被内核阻塞。这时第二个进程调用db_open，发现索引文件的长度为0，于是初始化空闲链表和散列链表，接着第二个进程向数据库中添加了一条记录。此时第二个进程被阻塞，第一个进程继续运行，它发现索引文件的长度为0（因为第一个进程调用fstat在前，然后第二个进程再初始化索引文件），所以第一个进程重新初始化空闲链表和散列链表，第二个进程写入的记录就被抹去了。避免发生这种情况的方法是进行加锁，为此可以使用14.3节中的readw_lock，writew_lock和un_lock这三个宏。

[131-137] 如果索引文件的长度是0，那么该文件刚刚被创建的，所以需要初始化它所包含

的空闲链表和散列链表指针。注意，用格式字符串%*d将数据库指针从整型变换为ASCII字符串。（在_db_writeidx和_db_writeptr中还将使用这种格式字符串。）这一格式告诉sprintf取PTR_SZ参数，用它作为下一个参数的最小字段宽度，在此处为0（因为正在创建一个新的数据库，所以这里使指针取初值为0）。其作用是强迫创建的字符串至少包含PTR_SZ个字符（在左边用空格填充）。在_db_writeidx和_db_writeptr中，将传送一个非0指针值，但是首先将验证指针值不大于PTR_MAX，以保证写入数据库的指针字符串恰好为PTR_SZ(6)个字符。

725

```
138             hash[0] = 0;
139             for (i = 0; i < NHASH_DEF + 1; i++)
140                 strcat(hash, asciiptr);
141             strcat(hash, "\n");
142             i = strlen(hash);
143             if (write(db->idxfd, hash, i) != i)
144                 err_dump("db_open: index file init write error");
145         }
146         if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147             err_dump("db_open: un_lock error");
148     }
149     db_rewind(db);
150     return(db);
151 }
152 /*
153  * Allocate & initialize a DB structure and its buffers.
154  */
155 static DB *
156 _db_alloc(int namelen)
157 {
158     DB      *db;
159     /*
160      * Use calloc, to initialize the structure to zero.
161      */
162     if ((db = calloc(1, sizeof(DB))) == NULL)
163         err_dump("_db_alloc: calloc error for DB");
164     db->idxfd = db->datfd = -1;             /* descriptors */
165     /*
166      * Allocate room for the name.
167      * +5 for ".idx" or ".dat" plus null at end.
168      */
169     if ((db->name = malloc(namelen + 5)) == NULL)
170         err_dump("_db_alloc: malloc error for name");
```

[138-151] 继续初始化新创建的数据库。构造散列表，将它写到索引文件中。然后，解锁索引文件，清除数据库文件指针，返回DB结构指针作为句柄，以便调用者以后用于其他数据库函数。

[152-164] db_open调用函数_db_alloc为DB结构分配空间，包括一个索引缓冲和一个数据缓冲。用calloc分配存储区，用以保存DB结构，并将该区各单元全部置初值为0。这产生了一个副作用，就是将数据库文件描述符也设置为0，因此需将它们重新设置为−1，以表示它们至此还不是有效的。

[165-170] 分配空间以存放数据库文件的名字。如db_open中所说明的那样，使用缓冲区

创建文件名，通过更改名字的后缀表示是索引文件还是数据文件。 726

```
171     /*
172      * Allocate an index buffer and a data buffer.
173      * +2 for newline and null at end.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: malloc error for index buffer");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: malloc error for data buffer");
179     return(db);
180   }
181   /*
182    * Relinquish access to the database.
183    */
184   void
185   db_close(DBHANDLE h)
186   {
187     _db_free((DB *)h);  /* closes fds, free buffers & struct */
188   }
189   /*
190    * Free up a DB structure, and all the malloc'ed buffers it
191    * may point to.  Also close the file descriptors if still open.
192    */
193   static void
194   _db_free(DB *db)
195   {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171-180] 为索引文件和数据文件的缓冲分配空间。索引缓冲和数据缓冲的大小在apue_db.h中定义。数据库函数库可以通过让这些缓冲按需要扩张来得到增强，其方法可以是记录这两个缓冲的大小，然后在需要更大的缓冲时调用realloc。最后，返回已分配到的DB结构的指针。

[181-188] db_close函数只是一个包装，它将数据库句柄转换为DB结构的指针，将其传送给_db_free函数，由该函数释放资源以及DB结构。

[189-199] db_open在打开索引文件和数据文件时如果发生错误，则调用_db_free释放资源；应用程序在结束对数据库的使用后，db_close也调用_db_free。如果数据库索引文件的文件描述符有效，那么关闭该文件；对数据文件的文件描述符也作同样处理。（回忆当在_db_alloc中分配一新的DB结构时，对每个文件描述符都赋初值−1。如果不能打开两个数据库文件中的一个，由于相应的文件描述符仍为−1，于是也就无需关闭它。） 727

```
200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207   }
```

```
208  /*
209   * Fetch a record.  Return a pointer to the null-terminated data.
210   */
211  char *
212  db_fetch(DBHANDLE h, const char *key)
213  {
214    DB      *db = h;
215    char    *ptr;

216    if (_db_find_and_lock(db, key, 0) < 0) {
217        ptr = NULL;             /* error, record not found */
218        db->cnt_fetcherr++;
219    } else {
220        ptr = _db_readdat(db);  /* return pointer to data */
221        db->cnt_fetchok++;
222    }

223    /*
224     * Unlock the hash chain that _db_find_and_lock locked.
225     */
226    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227        err_dump("db_fetch: un_lock error");
228    return(ptr);
229  }
```

[200-207] 接着，释放动态分配的缓冲。可以安全地将一个空指针传送给free函数，因此也就无需事先检查每个缓冲指针的值，但无论如何还是要这样做，因为只释放已分配的对象被认为是一种较好的编程风格。（并非所有释放函数都像free那样容忍差错。）最后，释放DB结构占用的存储区。

[208-218] 函数db_fetch根据给定的键来读取一条记录。它首先调用_db_find_and_lock在数据库中查找该记录。若不能找到该记录，则将返回值（ptr）设置为NULL，并将不成功的记录搜索计数值加1。因为从_db_find_and_lock返回时，数据库索引文件是加锁的，所以先要解锁，然后再返回。

[219-229] 如果找到了记录，调用_db_readdat读相应的数据记录，并将成功的记录搜索计数值加1。在返回前，调用un_lock对索引文件解锁，然后，返回所找到记录的指针（如果没有找到所需记录，则返回NULL）。

728

```
230  /*
231   * Find the specified record.  Called by db_delete, db_fetch,
232   * and db_store.  Returns with the hash chain locked.
233   */
234  static int
235  _db_find_and_lock(DB *db, const char *key, int writelock)
236  {
237    off_t   offset, nextoffset;

238    /*
239     * Calculate the hash value for this key, then calculate the
240     * byte offset of corresponding chain ptr in hash table.
241     * This is where our search starts.  First we calculate the
242     * offset in the hash table for this key.
243     */
244    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245    db->ptroff = db->chainoff;

246    /*
```

```
247      * We lock the hash chain here.  The caller must unlock it
248      * when done.  Note we lock and unlock only the first byte.
249      */
250     if (writelock) {
251         if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252             err_dump("_db_find_and_lock: writew_lock error");
253     } else {
254         if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255             err_dump("_db_find_and_lock: readw_lock error");
256     }
257     /*
258      * Get the offset in the index file of first record
259      * on the hash chain (can be 0).
260      */
261     offset = _db_readptr(db, db->ptroff);
```

[230-237] _db_find_and_lock函数在函数库内部用于按给定的键查找记录。在搜索记录时，如果想在索引文件上加一把写锁，则将writelock参数设置为非0值；如果将writelock参数设置为0，则在搜索记录时，在索引文件上加读锁。

[238-256] 在_db_find_and_lock中准备遍历散列链。将键变换为散列值，用其计算在文件中相应散列链的起始地址（chainoff）。在遍历散列链前，等待获得锁。注意，只锁该散列链开始处的第1个字节，这种方式允许多个进程同时搜索不同的散列链，因此增加了并发性。

[257-261] 调用_db_readptr读散列链中的第一个指针。如果该函数返回0，则该散列链为空。

729

```
262     while (offset != 0) {
263         nextoffset = _db_readidx(db, offset);
264         if (strcmp(db->idxbuf, key) == 0)
265             break;          /* found a match */
266         db->ptroff = offset; /* offset of this (unequal) record */
267         offset = nextoffset; /* next one to compare */
268     }
269     /*
270      * offset == 0 on error (record not found).
271      */
272     return(offset == 0 ? -1 : 0);
273 }
274 /*
275  * Calculate the hash value for a key.
276  */
277 static DBHASH
278 _db_hash(DB *db, const char *key)
279 {
280     DBHASH      hval = 0;
281     char        c;
282     int         i;
283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i;      /* ascii char times its 1-based index */
285     return(hval % db->nhash);
286 }
```

[262-268] while循环遍历散列链中的每一条索引记录，并比较键。调用函数_db_

readidx读取每条索引记录。它将当前记录的键填入DB结构中的idxbuf字段。如果_db_readidx返回0，则已到达散列链的最后一个记录项。

[269-273] 如果在循环后，offset为0，那么已到达散列链末端并且没有找到匹配键，于是返回−1；否则，找到了匹配记录（用break语句退出了循环），那么就返回0表示成功。此时，ptroff字段包含前一索引记录的地址，datoff包含数据记录的地址，datlen是数据记录的长度。当沿着散列链进行遍历时，必须始终跟踪当前索引记录的前一条索引记录，其中有一个指针指向当前索引记录。这一点在删除一条记录时很有用，因为必须修改当前索引记录的前一条记录的链表指针以删除当前记录。

[274-286] _db_hash根据给定的键计算散列值。它将键中的每一个ASCII字符乘以这个字符在字符串中以1开始的索引号，将这些结果加起来，除以散列表记录项数，将余数作为这个键的散列值。回忆散列表记录项数是137，它是一个素数，按Knuth [1998]，素数散列通常提供良好的分布特性。

730

```
287   /*
288    * Read a chain ptr field from anywhere in the index file:
289    * the free list pointer, a hash table chain ptr, or an
290    * index record chain ptr.
291    */
292   static off_t
293   _db_readptr(DB *db, off_t offset)
294   {
295     char    asciiptr[PTR_SZ + 1];

296     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297         err_dump("_db_readptr: lseek error to ptr field");
298     if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299         err_dump("_db_readptr: read error of ptr field");
300     asciiptr[PTR_SZ] = 0;       /* null terminate */
301     return(atol(asciiptr));
302   }

303   /*
304    * Read the next index record.  We start at the specified offset
305    * in the index file.  We read the index record into db->idxbuf
306    * and replace the separators with null bytes.  If all is OK we
307    * set db->datoff and db->datlen to the offset and length of the
308    * corresponding data record in the data file.
309    */
310   static off_t
311   _db_readidx(DB *db, off_t offset)
312   {
313     ssize_t             i;
314     char            *ptr1, *ptr2;
315     char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316     struct iovec    iov[2];
```

[287-302] _db_readptr函数读取以下三种不同链表指针中的任意一种：(1) 索引文件最开始处指向空闲链表中第一条索引记录的指针；(2) 散列表中指向散列链的第一条索引记录的指针；(3) 存放在每条索引记录开始处、指向下一条记录的指针（这里的索引记录既可以处于一条散列链中，也可以处于空闲链表中）。返回前，将指针从ASCII形式变换为长整型。此函数不进行任何加锁操作，所以其调用

者应事先做好必要的加锁。

[303-316] _db_readidx函数用于从索引文件的指定偏移量处读取索引记录。如果成功，该函数将返回链表中下一条记录的偏移量。并且该函数填充DB结构的许多字段：idxoff包含索引文件中当前记录的偏移量，ptrval包含在散列链表中下一条索引项的偏移量，idxlen包含当前索引记录的长度，idxbuf包含实际索引记录，datoff包含数据文件中该记录的偏移量，datlen包含该数据记录的长度。 731

```
317     /*
318      * Position index file and record the offset.  db_nextrec
319      * calls us with offset==0, meaning read from current offset.
320      * We still need to call lseek to record the current offset.
321      */
322     if ((db->idxoff = lseek(db->idxfd, offset,
323       offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readidx: lseek error");

325     /*
326      * Read the ascii chain ptr and the ascii length at
327      * the front of the index record.  This tells us the
328      * remaining size of the index record.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len  = PTR_SZ;
332     iov[1].iov_base = asciilen;
333     iov[1].iov_len  = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1);     /* EOF for db_nextrec */
337         err_dump("_db_readidx: readv error of index record");
338     }

339     /*
340      * This is our return value; always >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;         /* null terminate */
343     db->ptrval = atol(asciiptr); /* offset of next key in chain */

344     asciilen[IDXLEN_SZ] = 0;      /* null terminate */
345     if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346       db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readidx: invalid length");
```

[317-324] 按调用者提供的参数，查找索引文件偏移量。在DB结构中，记录该偏移量，为此即使调用者想要在当前文件偏移量处读记录（设置offset为0），仍需要调用lseek以确定当前偏移量。因为在索引文件中，索引记录决不会存放在偏移量为0处，所以可以放心地使用0表示“从当前偏移量处读”。

[325-338] 调用readv读在索引记录开始处的两个定长字段：指向下一条索引记录的链表指针和该索引记录余下部分的长度（余下部分是不定长的）。

[339-347] 变换下一记录的偏移量为整型，并存放到ptrval字段（这将作为此函数的返回值）。然后将索引记录的长度变换为整型，并存放在idxlen字段。 732

```
348     /*
349      * Now read the actual index record.  We read it into the key
350      * buffer that we malloced when we opened the database.
```

```
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: read error of index record");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
355         err_dump("_db_readidx: missing newline");
356     db->idxbuf[db->idxlen-1] = 0;    /* replace newline with null */
357     /*
358      * Find the separators in the index record.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: missing first separator");
362     *ptr1++ = 0;                /* replace SEP with null */
363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: missing second separator");
365     *ptr2++ = 0;                /* replace SEP with null */
366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: too many separators");
368     /*
369      * Get the starting offset and length of the data record.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: starting offset < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: invalid length");
375     return(db->ptrval);     /* return offset of next key in chain */
376 }
```

[348-356] 将索引记录的不定长部分读入DB结构中的idxbuf字段。该记录应以换行符结束，将该字符代换为NULL字节。如果索引文件已遭破坏，那么调用err_dump函数构造core文件后终止。

[357-367] 将索引记录划分成三个字段：键、对应数据记录的偏移量和数据记录的长度。strchr函数在给定字符串中找到首次出现的指定字符。这里要寻找的是记录中分隔字段的字符（SEP，将其定义为冒号）。

[368-376] 将数据记录的偏移量和长度变换为整型，并把它们存放在DB结构中。然后，返回散列链中下一条记录的偏移量。注意并不读数据记录，这由调用者自己完成。例如，在db_fetch中，在_db_find_and_lock按键找到索引记录前是不读取数据记录的。

733

```
377 /*
378  * Read the current data record into the data buffer.
379  * Return a pointer to the null-terminated data buffer.
380  */
381 static char *
382 _db_readdat(DB *db)
383 {
384     if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385         err_dump("_db_readdat: lseek error");
386     if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387         err_dump("_db_readdat: read error");
388     if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
389         err_dump("_db_readdat: missing newline");
390     db->datbuf[db->datlen-1] = 0; /* replace newline with null */
391     return(db->datbuf);     /* return pointer to data record */
```

```
392   }
393   /*
394    * Delete the specified record.
395    */
396   int
397   db_delete(DBHANDLE h, const char *key)
398   {
399     DB      *db = h;
400     int     rc = 0;         /* assume record will be found */
401     if (_db_find_and_lock(db, key, 1) == 0) {
402         _db_dodelete(db);
403         db->cnt_delok++;
404     } else {
405         rc = -1;            /* not found */
406         db->cnt_delerr++;
407     }
408     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409         err_dump("db_delete: un_lock error");
410     return(rc);
411   }
```

[377-392] 在datoff和datlen已获正确值后，_db_readdat函数将数据记录的内容读入DB结构中的datbuf字段指向的缓冲区。

[393-411] db_delete函数用于删除与给定键匹配的一条记录。调用_db_find_and_lock判断在数据库中该记录是否存在，如果存在，则调用_db_dodelete函数执行删除该记录的操作。_db_find_and_lock的第3个参数控制对散列链是加读锁，还是写锁。此处，因为可能执行更改该链表的操作，所以要加一把写锁。_db_find_and_lock返回时，这把锁仍旧存在，为此不管是否找到了所需的记录，都需要除去这把锁。

734

```
412   /*
413    * Delete the current record specified by the DB structure.
414    * This function is called by db_delete and db_store, after
415    * the record has been located by _db_find_and_lock.
416    */
417   static void
418   _db_dodelete(DB *db)
419   {
420     int     i;
421     char    *ptr;
422     off_t   freeptr, saveptr;
423     /*
424      * Set data buffer and key to all blanks.
425      */
426     for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427         *ptr++ = SPACE;
428     *ptr = 0;   /* null terminate for _db_writedat */
429     ptr = db->idxbuf;
430     while (*ptr)
431         *ptr++ = SPACE;
432     /*
433      * We have to lock the free list.
434      */
```

```
435     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436         err_dump("_db_dodelete: writew_lock error");

437     /*
438      * Write the data record with all blanks.
439      */
440     _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412-431] _db_dodelete函数执行从数据库中删除一条记录的所有操作。(该函数也可以由db_store调用)。此函数的大部分工作仅仅是更新两个链表：空闲链表以及与键对应的散列链。当一条记录被删除后，将其键和数据记录设为空。本节后面将提到的函数db_nextrec要用到这一点。

[432-440] 调用writew_lock对空闲链表加写锁，这样能防止两个进程同时删除不同散列链上的记录时产生相互影响，因为要将被删除的记录移到空闲链表上，这将改变空闲链表指针，而一次只能有一个进程能这样做。

_db_dodelete调用函数_db_writedat清空数据记录，注意此时_db_writedat无需对数据文件加写锁，因为db_delete对这条记录的散列链已经加了写锁，这便保证不再会有其他进程能够读写该记录。

735

```
441     /*
442      * Read the free list pointer.  Its value becomes the
443      * chain ptr field of the deleted index record.  This means
444      * the deleted record becomes the head of the free list.
445      */
446     freeptr = _db_readptr(db, FREE_OFF);

447     /*
448      * Save the contents of index record chain ptr,
449      * before it's rewritten by _db_writeidx.
450      */
451     saveptr = db->ptrval;

452     /*
453      * Rewrite the index record.  This also rewrites the length
454      * of the index record, the data offset, and the data length,
455      * none of which has changed, but that's OK.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

458     /*
459      * Write the new free list pointer.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);

462     /*
463      * Rewrite the chain ptr that pointed to this record being
464      * deleted.  Recall that _db_find_and_lock sets db->ptroff to
465      * point to this chain ptr.  We set this chain ptr to the
466      * contents of the deleted record's chain ptr, saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: un_lock error");
471 }
```

[441-461] 读空闲链表指针，接着修改索引记录，让这条记录中的下一条记录指针指向空闲链表的第一条记录（如果空闲链表为空，则这个链表指针置为0)。既然已经清除

了键，就用正被删除的索引记录的偏移量更新空闲链表指针，也就是使其指向当前删除的这条记录，这样就将这条删除的记录加到了空闲链表之首。空闲链表实际上很像一个后进先出的堆栈（虽然是以首次适应算法分配空闲链表项）。

没有为每个文件分别设置空闲链表。当把一个删除的索引记录加入空闲链表时，该索引记录仍指向已删除的数据记录。有更好的处理方法，但复杂性增加了。

[462-471] 修改散列链中前一条记录的指针，使其指向正删除记录之后的一条记录，这样便从散列链中撤除了要删去的记录。最后对空闲链表解锁。 736

```
472   /*
473    * Write a data record.  Called by _db_dodelete (to write
474    * the record with blanks) and db_store.
475    */
476   static void
477   _db_writedat(DB *db, const char *data, off_t offset, int whence)
478   {
479     struct iovec    iov[2];
480     static char     newline = NEWLINE;

481     /*
482      * If we're appending, we have to lock before doing the lseek
483      * and write to make the two an atomic operation.  If we're
484      * overwriting an existing record, we don't have to lock.
485      */
486     if (whence == SEEK_END) /* we're appending, lock entire file */
487         if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488             err_dump("_db_writedat: writew_lock error");

489     if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490         err_dump("_db_writedat: lseek error");
491     db->datlen = strlen(data) + 1;  /* datlen includes newline */

492     iov[0].iov_base = (char *) data;
493     iov[0].iov_len  = db->datlen - 1;
494     iov[1].iov_base = &newline;
495     iov[1].iov_len  = 1;
496     if (writev(db->datfd, &iov[0], 2) != db->datlen)
497         err_dump("_db_writedat: writev error of data record");

498     if (whence == SEEK_END)
499         if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500             err_dump("_db_writedat: un_lock error");
501   }
```

[472-491] 调用函数_db_writedat写一个数据记录。当删除一条记录时，调用函数_db_writedat清空数据记录，此时_db_writedat并不对数据文件加写锁，因为db_delete对该记录的散列链已经加了写锁，这便保证不再会有其他进程能够读写这条记录。在本节稍后处解说db_store函数时，会遇到_db_writedat函数追加数据文件的情况，此时就必须对该文件加锁。

确定要写数据记录的位置。要写的字节数是记录长度+1字节，这1个字节是为表示记录终止的换行符而增加的。

[492-501] 设置iovec数组，调用writev写数据记录和换行符。因为不能想当然地认为调用者缓冲区的尾端有空间可以加换行符，所以先将换行符送入另一个缓冲，然后再从该缓冲写至数据记录。如果正对文件添加一条记录，则释放早先获得的锁。 737

```
502  /*
503   * Write an index record.  _db_writedat is called before
504   * this function to set the datoff and datlen fields in the
505   * DB structure, which we need to write the index record.
506   */
507  static void
508  _db_writeidx(DB *db, const char *key,
509               off_t offset, int whence, off_t ptrval)
510  {
511    struct iovec    iov[2];
512    char            asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
513    int             len;
514    char            *fmt;

515    if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
516        err_quit("_db_writeidx: invalid ptr: %d", ptrval);
517    if (sizeof(off_t) == sizeof(long long))
518        fmt = "%s%c%lld%c%d\n";
519    else
520        fmt = "%s%c%ld%c%d\n";
521    sprintf(db->idxbuf, fmt, key, SEP, db->datoff, SEP, db->datlen);
522    if ((len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
523        err_dump("_db_writeidx: invalid length");
524    sprintf(asciiptrlen, "%*ld%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

525    /*
526     * If we're appending, we have to lock before doing the lseek
527     * and write to make the two an atomic operation.  If we're
528     * overwriting an existing record, we don't have to lock.
529     */
530    if (whence == SEEK_END)     /* we're appending */
531        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
532          SEEK_SET, 0) < 0)
533            err_dump("_db_writeidx: writew_lock error");
```

[502-524] 调用_db_writeidx函数写一条索引记录。在验证散列链中下一个指针有效后，创建索引记录，并将它的后半部分存放到idxbuf中。需要索引记录这一部分的长度以创建该记录的前半部分，而前半部分被存放到局部变量asciiptrlen中。注意，基于off_t数据类型的长度，选择传送给sprintf的格式字符串。即使32位的系统也能提供64位的文件偏移量，所以不能假定off_t数据类型的长度。

[525-533] 和_db_writedat一样，只有在向索引文件添加新索引记录时这一函数才需要加锁。_db_dodelete调用此函数是为了重写一条现有的索引记录，在这种情况下调用者已经在散列链上加了写锁，所以不再需要加另外的锁。

738

```
534    /*
535     * Position the index file and record the offset.
536     */
537    if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
538        err_dump("_db_writeidx: lseek error");

539    iov[0].iov_base = asciiptrlen;
540    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
541    iov[1].iov_base = db->idxbuf;
542    iov[1].iov_len  = len;
543    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
544        err_dump("_db_writeidx: writev error of index record");
```

```
545     if (whence == SEEK_END)
546         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
547           SEEK_SET, 0) < 0)
548             err_dump("_db_writeidx: un_lock error");
549   }

550   /*
551    * Write a chain ptr field somewhere in the index file:
552    * the free list, the hash table, or in an index record.
553    */
554   static void
555   _db_writeptr(DB *db, off_t offset, off_t ptrval)
556   {
557     char    asciiptr[PTR_SZ + 1];

558     if (ptrval < 0 || ptrval > PTR_MAX)
559         err_quit("_db_writeptr: invalid ptr: %d", ptrval);
560     sprintf(asciiptr, "%*ld", PTR_SZ, ptrval);

561     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
562         err_dump("_db_writeptr: lseek error to ptr field");
563     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
564         err_dump("_db_writeptr: write error of ptr field");
565   }
```

[534-549] 设置索引文件偏移量，从此处开始写索引记录，将该偏移量存入DB结构的idxoff字段。因为是在两个分开的缓冲中构造索引记录，所以调用writev将它存放到索引文件中。如果是追加该文件，则释放在定位操作前加的锁。从并发运行的进程添加新记录至同一数据库角度思考问题，那么这把锁使定位（seek）和写成为原子操作。

[550-565] _db_writeptr用于将一个链表指针写至索引文件中。验证该指针在索引文件的边界范围内，然后将它变换成ASCII字符串。按指定的偏移量在索引文件中定位，接着将该指针ASCII字符串写入索引文件。 739

```
566   /*
567    * Store a record in the database.  Return 0 if OK, 1 if record
568    * exists and DB_INSERT specified, -1 on error.
569    */
570   int
571   db_store(DBHANDLE h, const char *key, const char *data, int flag)
572   {
573     DB      *db = h;
574     int     rc, keylen, datlen;
575     off_t   ptrval;

576     if (flag != DB_INSERT && flag != DB_REPLACE &&
577       flag != DB_STORE) {
578         errno = EINVAL;
579         return(-1);
580     }
581     keylen = strlen(key);
582     datlen = strlen(data) + 1;      /* +1 for newline at end */
583     if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
584         err_dump("db_store: invalid data length");

585     /*
586      * _db_find_and_lock calculates which hash table this new record
587      * goes into (db->chainoff), regardless of whether it already
```

```
588         * exists or not. The following calls to _db_writeptr change the
589         * hash table entry for this chain to point to the new record.
590         * The new record is added to the front of the hash chain.
591         */
592        if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
593            if (flag == DB_REPLACE) {
594                rc = -1;
595                db->cnt_storerr++;
596                errno = ENOENT;     /* error, record does not exist */
597                goto doreturn;
598            }
```

[566-584] db_store函数用于将一条记录加到数据库中。首先验证参数flag的值；然后，查明数据记录长度是否有效，如果无效，则构造core文件并退出。作为一个例子这样处理无可厚非，但如果构造的是可正式应用的函数库，那么最好返回出错状态而非退出，这样可以给应用程序一个恢复机会。

[585-598] 调用_db_find_and_lock以查看这个记录是否已经存在。如果记录并不存在且指定的标志为DB_INSERT或DB_STORE，或者记录存在且指定的标志为DB_REPLACE或DB_STORE，那么这些都是允许的。替换一条已存在的记录，指的是该记录的键一样，而数据记录很可能不一样。注意，因为db_store很可能会修改散列链，所以用_db_find_and_lock的最后一个参数指明要对散列链加写锁。

740

```
599            /*
600             * _db_find_and_lock locked the hash chain for us; read
601             * the chain ptr to the first index record on hash chain.
602             */
603            ptrval = _db_readptr(db, db->chainoff);

604            if (_db_findfree(db, keylen, datlen) < 0) {
605                /*
606                 * Can't find an empty record big enough. Append the
607                 * new record to the ends of the index and data files.
608                 */
609                _db_writedat(db, data, 0, SEEK_END);
610                _db_writeidx(db, key, 0, SEEK_END, ptrval);

611                /*
612                 * db->idxoff was set by _db_writeidx.  The new
613                 * record goes to the front of the hash chain.
614                 */
615                _db_writeptr(db, db->chainoff, db->idxoff);
616                db->cnt_stor1++;
617            } else {
618                /*
619                 * Reuse an empty record. _db_findfree removed it from
620                 * the free list and set both db->datoff and db->idxoff.
621                 * Reused record goes to the front of the hash chain.
622                 */
623                _db_writedat(db, data, db->datoff, SEEK_SET);
624                _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
625                _db_writeptr(db, db->chainoff, db->idxoff);
626                db->cnt_stor2++;
627            }
```

[599-603] 在调用_db_find_and_lock后，程序分成四种情况。前两种情况中，没有找到相应的记录，所以添加新记录。读散列链上第一项的偏移量。

[604-616] 第一种情况：调用_db_findfree在空闲链表中搜索一条已删除的记录，它的键长度和数据长度与参数keylen，和datlen相同。如果没有找到对应大小的记录。这意味着要将这条新记录添加到索引文件和数据文件的末尾。调用_db_writedat写数据部分，调用_db_writeidx写索引部分，调用_db_writeptr将新记录加到对应的散列链的链首。将对此种情况的执行进行计数的计数器（cnt_stor1）值加1，以便观察数据库的运行状况。

[617-627] 第二种情况：_db_findfree找到了对应大小的空记录，并将这条空记录从空闲链表上移下来（很快就会见到_db_findfree的实现），写入新的索引记录和数据记录，然后，与第一种情况一样，将新记录加到对应的散列链的链首。将对此种情况的执行进行计数的计数器（cnt_stor2）值加1，以便观察数据库的运行状况。

741

```
628     } else {                        /* record found */
629         if (flag == DB_INSERT) {
630             rc = 1;     /* error, record already in db */
631             db->cnt_storerr++;
632             goto doreturn;
633         }

634         /*
635          * We are replacing an existing record.  We know the new
636          * key equals the existing key, but we need to check if
637          * the data records are the same size.
638          */
639         if (datlen != db->datlen) {
640             _db_dodelete(db);   /* delete the existing record */

641             /*
642              * Reread the chain ptr in the hash table
643              * (it may change with the deletion).
644              */
645             ptrval = _db_readptr(db, db->chainoff);

646             /*
647              * Append new index and data records to end of files.
648              */
649             _db_writedat(db, data, 0, SEEK_END);
650             _db_writeidx(db, key, 0, SEEK_END, ptrval);

651             /*
652              * New record goes to the front of the hash chain.
653              */
654             _db_writeptr(db, db->chainoff, db->idxoff);
655             db->cnt_stor3++;
656         } else {
```

[628-633] 另两种情况是具相同键的记录在数据库中已存在。如果不想替换该记录，则设置表示一条记录已经存在的返回码，将对存储出错计数的计数器（cnt_storerr）值加1，然后跳转至函数末尾，在此处理公共返回逻辑。

[634-656] 第三种情况：要替换一条现存记录，而新数据记录的长度与已存在记录的长度不一样。调用_db_dodelete将老记录删除，该删除记录将放在空闲链表的链

首。然后，调用_db_writedat和_db_writeidx将新记录添加到索引文件和数据文件的末尾（也可以用其他方法，如可以再找一找是否有数据大小正好的已删除记录项）。最后调用_db_writeptr将新记录加到对应的散列链的链首。
742 DB结构中的cnt_stor3计数器记录发生此种情况的次数。

```
657             /*
658              * Same size data, just replace data record.
659              */
660             _db_writedat(db, data, db->datoff, SEEK_SET);
661             db->cnt_stor4++;
662         }
663     }
664     rc = 0;     /* OK */
665 doreturn: /* unlock hash chain locked by _db_find_and_lock */
666     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
667         err_dump("db_store: un_lock error");
668     return(rc);
669 }
670 /*
671  * Try to find a free index record and accompanying data record
672  * of the correct sizes.  We're only called by db_store.
673  */
674 static int
675 _db_findfree(DB *db, int keylen, int datlen)
676 {
677     int     rc;
678     off_t   offset, nextoffset, saveoffset;
679     /*
680      * Lock the free list.
681      */
682     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
683         err_dump("_db_findfree: writew_lock error");
684     /*
685      * Read the free list pointer.
686      */
687     saveoffset = FREE_OFF;
688     offset = _db_readptr(db, saveoffset);
```

[657-663] 第四种情况：要替换一条现存记录，而新数据记录的长度与已存在的记录的长度恰好一样。这是最容易的情况，只需要重写数据记录即可，并将计数器（cnt_stor4）的值加1。

[664-669] 在正常情况下，设置表示成功的返回码，然后进入公共返回逻辑。对散列链解锁（这把锁是由调用_db_find_and_lock而加上的），然后返回调用者。

[670-688] _db_findfree函数试图找到一个指定大小的空闲索引记录和相关联的数据记录。_db_findfree需要对空闲链表加写锁以避免与其他使用空闲链表的进程
743 互相干扰。在对空闲链表加写锁后，得到空闲链表链首的指针地址。

```
689     while (offset != 0) {
690         nextoffset = _db_readidx(db, offset);
691         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
692             break;      /* found a match */
```

```
693             saveoffset = offset;
694             offset = nextoffset;
695         }
696         if (offset == 0) {
697             rc = -1;    /* no match found */
698         } else {
699             /*
700              * Found a free record with matching sizes.
701              * The index record was read in by _db_readidx above,
702              * which sets db->ptrval.  Also, saveoffset points to
703              * the chain ptr that pointed to this empty record on
704              * the free list.  We set this chain ptr to db->ptrval,
705              * which removes the empty record from the free list.
706              */
707             _db_writeptr(db, saveoffset, db->ptrval);
708             rc = 0;
709             /*
710              * Notice also that _db_readidx set both db->idxoff
711              * and db->datoff.  This is used by the caller, db_store,
712              * to write the new index record and data record.
713              */
714         }
715         /*
716          * Unlock the free list.
717          */
718         if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
719             err_dump("_db_findfree: un_lock error");
720         return(rc);
721     }
```

[689-695] _db_findfree中的while循环遍历空闲链表，以搜寻一个有匹配键长度和数据长度的记录项。在这个简单的实现中，只有当一个已删除记录的键长度及数据长度与要加入的新记录的键长度及数据长度一样时，才重用已删除记录的空间。其他更好的重用删除空间的方法一般更复杂。

[696-714] 如果找不到具有所要求的键长度和数据长度的可用记录，则设置表示失败的返回码。否则，将已找到记录的下一个链表指针值写至前一记录的链表指针，这样就从空闲链表中移除了该记录。

[715-721] 一旦结束对空闲链表的操作，就释放写锁，然后对调用者返回状态码。 744

```
722     /*
723      * Rewind the index file for db_nextrec.
724      * Automatically called by db_open.
725      * Must be called before first db_nextrec.
726      */
727     void
728     db_rewind(DBHANDLE h)
729     {
730         DB      *db = h;
731         off_t   offset;
732         offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */
733         /*
734          * We're just setting the file offset for this process
```

```
735      * to the start of the index records; no need to lock.
736      * +1 below for newline at end of hash table.
737      */
738     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
739         err_dump("db_rewind: lseek error");
740 }
741 /*
742  * Return the next sequential record.
743  * We just step our way through the index file, ignoring deleted
744  * records.  db_rewind must be called before this function is
745  * called the first time.
746  */
747 char *
748 db_nextrec(DBHANDLE h, char *key)
749 {
750     DB      *db = h;
751     char    c;
752     char    *ptr;
```

[722-740] db_rewind函数用于把数据库重置到“起始状态”，将索引文件的文件偏移量定位在索引文件的第一条索引记录（紧跟在散列表之后）。（回忆图20-1中索引文件的结构。）

[741-752] db_nextrec函数返回数据库的下一条记录，返回值是指向数据缓冲的指针。如果调用者提供的Key参数非空，那么相应键复制到该缓冲中。调用者负责分配可以存放键的足够大的缓冲。大小为IDXLEN_MAX字节的缓冲足够存放任一键。记录按它们在数据库文件中存放的顺序逐一返回，因此，记录并不按键值的大小排序。另外，db_nextrec并不跟随散列链表，所以也可能读到已删除的记录，只是不向调用者返回这种已删除记录。

745

```
753     /*
754      * We read lock the free list so that we don't read
755      * a record in the middle of its being deleted.
756      */
757     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
758         err_dump("db_nextrec: readw_lock error");
759     do {
760         /*
761          * Read next sequential index record.
762          */
763         if (_db_readidx(db, 0) < 0) {
764             ptr = NULL;     /* end of index file, EOF */
765             goto doreturn;
766         }
767         /*
768          * Check if key is all blank (empty record).
769          */
770         ptr = db->idxbuf;
771         while ((c = *ptr++) != 0  &&  c == SPACE)
772             ;   /* skip until null byte or nonblank */
773     } while (c == 0);   /* loop until a nonblank key is found */
774     if (key != NULL)
775         strcpy(key, db->idxbuf);    /* return key */
```

```
776     ptr = _db_readdat(db);  /* return pointer to data buffer */
777     db->cnt_nextrec++;

778   doreturn:
779     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
780         err_dump("db_nextrec: un_lock error");
781     return(ptr);
782   }
```

[753-758] 首先，需要对空闲链表加读锁，使得正在读该链表时，其他进程不能从中删除某一记录。

[759-773] 调用_db_readidx读下一条记录。传送给该函数的偏移量参数值为0，以此通知该函数从当前位置继续读索引记录。因为是逐条顺序地读索引文件，所以可能会读到已删除的记录。仅需返回有效记录，所以跳过键是全空格的记录（回忆_db_dodelete函数以设置为全空格方式清除键）。

[774-782] 当找到一个有效键时，如果调用者已提供缓冲，则将该键复制至该缓冲。然后读数据记录，并将返回值设置为指向包含数据记录的内部缓冲的指针值。使统计计数器值加1，对空闲链表解锁，最后返回指向数据记录的指针。

746

通常在下列形式的循环中的使用db_rewind和db_nextrec这两个函数：

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

前面曾警告过，记录的返回没有一定的次序，它们并不按键的顺序返回。

如果db_nextrec函数在循环中被调用时数据库正被修改，则db_nextrec返回的记录只是变化中的数据库在某一时间点的快照（snapshot）。db_nextrec被调用时总是返回一条“正确”的记录，也就是说它不会返回一条已删除的记录；但有可能一条记录刚被db_nextrec返回后就被删除；类似地，如果db_nextrec刚跳过一条已删除的记录，这条记录的空间就被一条新记录重用，此时除非用db_rewind并重新遍历一遍，否则将看不到这条新的记录。如果通过db_nextrec获得一份数据库的准确的“冻结”的快照很重要，则应做到在这段时间内没有添加和删除。

下面来看db_nextrec使用的加锁。因为并不使用任何的散列链表，也不能判断每条记录属于哪条散列链，所以有可能当db_nextrec读取一条记录时，其索引记录正在被删除。为了防止这种情况，db_nextrec对空闲链表加读锁，这样就可避免与_db_dodelete和_db_findfree相互影响。

在结束对db.c源文件的解释说明之前，还需要说明在向文件的末尾添加索引记录或数据记录时，需要加锁。在第1情况和第3种情况中，db_store调用_db_writeidx和_db_writedat时，第3个参数为0，第4个参数为SEEK_END。这里，第4个参数作为一个标志用来告诉这两个函数，新的记录将被添加到文件的末尾。_db_writeidx用到的技术是对索引文件加写锁，加锁的范围从散列链的末尾到文件的末尾。这不会影响其他数据库的读用户和写用户（这些用户将对散列链加锁），但如果其他用户此时调用db_store来添加数据则会被阻止。_db_writedat使用的方法是对整个数据文件加写锁。同样这也不会影响其他数据库的读用户和写用户（它们甚至不对数据文件加锁），但如果其他用户此时调用db_store来向数据文件添加数据则会被阻止（见习题20.3）。

20.9　性能

为了测试这一数据库函数库，也为了获得一些与典型应用的数据访问模式有关的时间测试数据，我们编写了一个测试程序。该程序接受两个命令行参数：要创建的子进程的个数以及每个子进程向数据库写的数据库记录的条数（*nrec*），然后创建一个空的数据库（通过调用db_open），通过forks创建指定数目的子进程，并等待所有子进程结束。每个子进程执行以下
747 步骤：

(1) 向数据库写*nrec*条记录。

(2) 通过键值读回*nrec*条记录。

(3) 执行下面的循环*nrec* × 5次：

(a) 随机读一条记录。

(b) 每循环37次，随机删除一条记录。

(c) 每循环11次，添加一条新记录并读回这条记录。

(d) 每循环17次，随机替换一条记录为新记录。在连续两次替换中，一次用同样大小的记录替换，一次用比以前更长的记录替换。

(4) 将此子进程写的所有记录删除。每删除一条记录，随机地寻找10条记录。

随着函数的调用次数增加，DB结构的cnt_xxx变量记录对数据库进行的操作数。每个子进程的操作数一般都会与其他子进程不一样，因为每个子进程用来选择记录的随机数生成器是根据其进程ID来初始化的。当*nrec*为500时，每个子进程的较典型的操作计数见表20-2。

表20-2　nrec为500时，每个子进程执行的操作的典型计数

操　作	计　数
db_store，DB_INSERT，无空白记录，添加	678
db_store，DB_INSERT，重用空白记录	164
db_store，DB_REPLACE，数据长度不同，添加	97
db_store，DB_REPLACE，数据长度相同	109
db_store，没有找到记录	19
db_fetch，找到记录	8 114
db_fetch，没有找到记录	732
db_delete，找到记录	842
db_delete，没有找到记录	110

读取的次数大约是存储或删除的10倍，这可能是许多数据库应用程序的典型情况。

每一个子进程只对该子进程所写的记录执行这些操作（读取、存储和删除）。由于所有的子进程对同一个数据库进行操作（虽然对不同的记录），所以会使用并发控制。数据库中的记录总条数与子进程数成比例增加。（当只有一个子进程时，一开始有*nrec*条记录写入数据库；当有两个子进程时，一开始有*nrec* × 2条记录写入数据库，依此类推。）

通过运行测试程序的三个不同版本来比较加粗锁和加细锁提供的并发，并且比较三种不同的加锁方式（不加锁、建议性锁和强制性锁）。第一个版本加细锁，用20.8节中的源代码，曾将
748 此称为细锁版本；第二个版本通过改变加锁调用而使用粗锁，20.6节对此已介绍过；第三个版本将所有加锁调用均去掉，这样可以计算加锁的开销。通过改变数据库文件的权限标志位，还可以使第一个版本和第二个版本（加细锁和加粗锁）使用建议性锁或强制性锁（本节所有的测

试中，仅对加细锁的实现测量了采用强制性锁的时间）。

本节所有的测试都在一台运行Solaris 9的SPARC系统上进行。

1. 单进程的结果

表20-3显示了只有一个子进程运行时的结果，*nrec*分别为500、1 000和2 000。

表20-3　单子进程、不同的nrec和不同的加锁方法

nrec	不加锁			建议性锁						强制性锁		
				粗锁			细锁			细锁		
	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock
500	0.42	0.89	1.31	0.42	1.17	1.59	0.41	1.04	1.45	0.46	1.49	1.95
1 000	1.51	3.89	5.41	1.64	4.13	5.78	1.63	4.12	5.76	1.73	6.34	8.07
2 000	3.91	10.06	13.98	4.09	10.30	14.39	4.03	10.63	14.66	4.47	16.21	20.70

最后12列显示的是以秒为单位的时间。在所有的情况下，用户CPU时间加上系统CPU时间都近似地等于时钟时间。这一组测试受CPU限制而不是受磁盘操作限制。

中间6列（建议性锁）对加粗锁和加细锁的结果基本一样。这是可以理解的，因为对于单个进程来说加粗锁和加细锁并没有区别。

比较不加锁和加建议性锁，可以看到加锁调用在系统CPU时间上增加了2%到31%。即使这些锁实际上并没有使用过（由于只有一个进程），`fcntl`系统调用仍会有一些时间的开销。另外，注意到用户CPU时间对于四种不同的加锁方案基本上一样，这是因为用户代码基本上是一样的（除了调用`fcntl`的次数有所不同外）。

关于表20-3要注意的最后一点是强制性锁比建议性锁增加了大约43%到54%的系统CPU时间。由于对加强制细锁和加建议细锁的加锁调用次数是一样的，故增加的系统开销来自读和写。

最后的测试是尝试有多个子进程的不加锁的程序。与预想的一样，结果是随机的错误。一般情况包括：加入到数据库中的记录找不到，测试程序异常退出等。几乎每次运行测试程序，就有不同的错误发生。这是典型的竞争状态—多个进程在没有任何加锁的情况下修改同一个文件，错误情况不可预测。

749

2. 多进程的结果

下一组测试主要查看粗锁和细锁的不同。前面说过，由于加细锁时数据库的各个部分被其他进程锁住的时间比加粗锁少，所以凭直觉加细锁应能够提供更好的并发性。表20-4显示了对*nrec*取500、子进程数目从1到12的测试结果。

所有的用户时间、系统时间和时钟时间的单位均为秒，所有这些时间均是父进程与所有子进程的总和。关于这些数据有许多需要考虑。

第8列（标记为“Δ Clock”）是加建议粗锁与加建议细锁的时钟时间的时间差，从该度量中可以看出使用细锁得到了多大的并发度。在运行测试程序的系统上，当并发进程数不多于7时，加粗锁与加细锁的效果大致相当；即使多于7个进程，使用细锁的时间减少也不大（一般少于3%），这不禁让人怀疑使用额外的代码来实现细锁是否值得。

希望从粗锁到细锁时钟时间会减少，最后也确实如此，但也认为就系统时间而言，对任何进程数，细锁将保持比粗锁高。这样预想的原因是对细锁调用了更多次的`fcntl`。如果将表20-2中的`fcntl`调用次数加起来，平均对粗锁有21 730次，细锁25 292次（表20-2中的每个操作对于粗锁要调用两次`fcntl`，而对于细锁前三个`db_store`及记录删除（记录找到）需要调用

四次fcntl)。基于此，认为由于增加了16%的fcntl调用次数，所以会增加细锁的系统时间。
750 然而，在测试中当进程数超过7后，加细锁的系统时间反而稍有下降，这不免让人迷惑。

表20-4　nrec=500时，不同加锁方法的比较

#Proc	建议性锁							强制性锁			
	粗锁			细锁			Δ	细锁			Δ
	User	Sys	Clock	User	Sys	Clock	Clock	User	Sys	Clock	Percent
1	0.41	1.00	1.42	0.41	1.05	1.47	0.05	0.47	1.40	1.87	33
2	1.10	2.81	3.92	1.11	2.80	3.92	0.00	1.15	4.06	5.22	45
3	2.17	5.27	7.44	2.19	5.18	7.37	−0.07	2.31	7.67	9.99	48
4	3.36	8.55	11.91	3.26	8.67	11.94	0.03	3.51	12.69	16.20	46
5	4.72	13.08	17.80	4.99	12.64	17.64	−0.16	4.91	19.21	24.14	52
6	6.45	17.96	24.42	6.83	17.29	24.14	−0.28	7.03	26.59	33.66	54
7	8.46	23.12	31.62	8.67	22.96	31.65	0.03	9.25	35.47	44.74	54
8	10.83	29.68	40.55	11.00	29.39	40.41	−0.14	11.67	45.90	57.63	56
9	13.35	36.81	50.23	13.43	36.28	49.76	−0.47	14.45	58.02	72.49	60
10	16.35	45.28	61.66	16.09	44.10	60.23	−1.43	17.43	70.90	88.37	61
11	18.97	54.24	73.24	19.13	51.70	70.87	−2.37	20.62	84.98	105.69	64
12	22.92	63.54	86.51	22.94	61.28	84.29	−2.22	24.41	101.68	126.20	66

对此作进一步分析，发现减少的原因是：对粗锁而言，保持这种锁的时间会比较长，于是增加了其他进程因这种锁而阻塞的可能性；而对细锁而言，由于加锁的时间较短，于是进程为此而阻塞的机会也就比较少。如果分析运行12个数据库进程的系统行为，将会看到加粗锁时的进程切换次数是加细锁时的3倍。这就意味着在使用细锁时，进程在锁上阻塞的机会要少得多。

最后一列（标记为“Δ percent”）是从加建议细锁到加强制细锁的系统CPU时间的百分比增量。这与在表20-3中看到的强制性锁显著增加（约33%到66%）系统时间是一致的。

由于所有这些测试的用户代码几乎一样（对加建议细锁和强制细锁增加了一些fcntl调用），预期对每一行的用户CPU时间应基本一样。

表20-4的第一行与表20-3中的*nrec*取500的那一行很相似。这与预期相一致。

图20-4是表20-4中加建议细锁的数据图。图中绘制了进程数从1到12的时钟时间，也绘制了用户CPU时间除以进程数后的每进程用户CPU时间，另外还绘制了系统CPU时间除以进程数后
751 的每进程系统CPU时间。

注意到这两个每进程CPU时间都是线性的，但时钟时间是非线性的。可能的原因是：当进程数增大时，操作系统用于进行进程切换的CPU时间增多。操作系统的开销是使时钟时间增加，但不会影响单个进程的CPU时间。

用户CPU时间随进程数增加的原因可能是因为数据库中有了更多的记录，每一条散列链更长，所以_db_find_and_lock函数平均要运行更长时间来找到一条记录。

20.10　小结

本章详细介绍了一个数据库函数库的设计与实现。考虑到篇幅，使这个函数库尽可能小和简单，但也包括了多进程并发控制需要的对记录加锁的功能。

此外，还使用不同数目的进程，以及不同的加锁方法：不加锁、建议性锁（细锁和粗锁）和强制性锁，研究了这个函数库的性能。可以看到加建议性锁比不加锁在时钟时间上增加了约10%，加强制性锁比建议性锁耗时再增加约33%至66%。

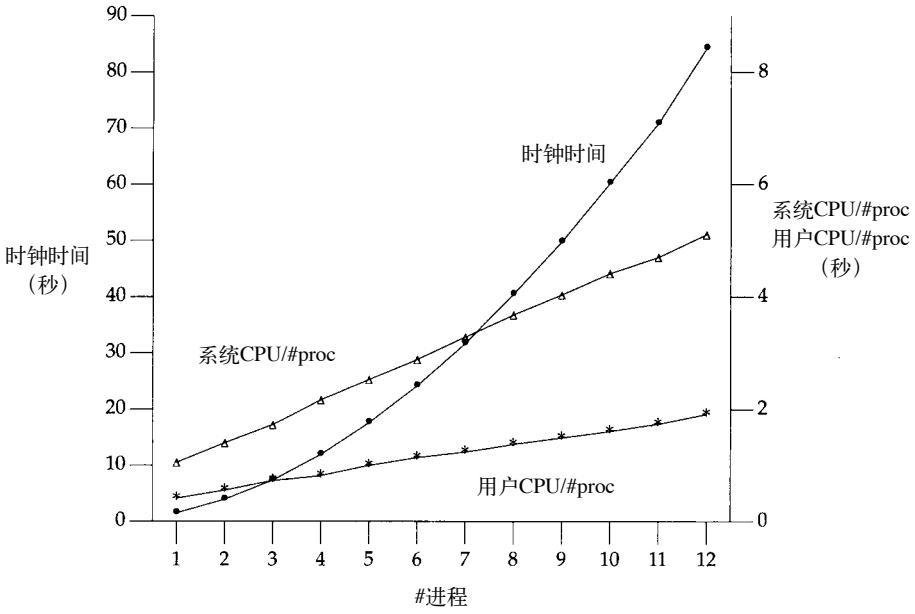

图20-4　表20-4中使用建议细锁的数据

习题

20.1 在_db_dodelete中使用的加锁是比较保守的。例如，如果等到真正要用空闲链表时再加锁，则可获得更大的并发度。如果将调用writew_lock移到调用_db_writedat和_db_readptr之间会发生什么呢？

20.2 如果db_nextrec不对空闲链表加读锁而它所读的记录正在被删除，描述在怎样的情况下，db_nextrec会返回一个正确的键但是数据记录却是空的（提示:查看_db_dodelete）。

20.3 在20.8节的结尾部分，描述了_db_writeidx和_db_writedat的加锁，曾说过这种加锁不会干涉除了调用db_store外的其他的读进程和写进程。如果改为强制性锁，这还成立吗？

20.4 怎样把fsync集成到这个数据库函数库中？

20.5 在db_store中，先写数据记录，然后再写索引记录。如果将次序颠倒，会发生什么？

20.6 建立一个新的数据库并写入一些记录。写一个程序调用db_nextrec来读数据库中的每条记录，并调用_db_hash来计算每条记录的散列值。根据每条散列链上的记录数画出直方图。_db_hash中的散列函数是否适当？

20.7 修改数据库函数，使得索引文件中散列链的数目可以在数据库建立时指定。

20.8 比较两种情况下数据库函数的性能：(a) 数据库与测试程序在同一台机器上；(b) 数据库与测试程序在不同的机器上，经由NFS进行访问。这个数据库函数库提供的记录锁机制还能工作吗？